プログラミング言語を作ろう

～関数型パーサ編～

A:Mac合宿 2022/2/19-20

前回までのあらすじ(1)

- 代数データ型
  - 直和型/直積型
  - 再帰的な型
- 関数
  - 関数定義
  - パターンマッチ
  - 再帰関数

前回までのあらすじ(2)

```haskell
data Shape = Circle Double -- Diameter
           | Rectangle Double Double -- Width, Height
           | Line Double Double -- Angle, Length

area :: Shape -> Double
area (Circle diameter) = radius * radius * 3.14
    where radius = diameter / 2
area (Rectangle width height) = width * height
area (Line _ _) = 0
```

```haskell
data IntList = Node Int IntList | IntNil

sum :: IntList -> Int
sum IntINil = 0
sum (Node value next) = value + (Sum next)
```

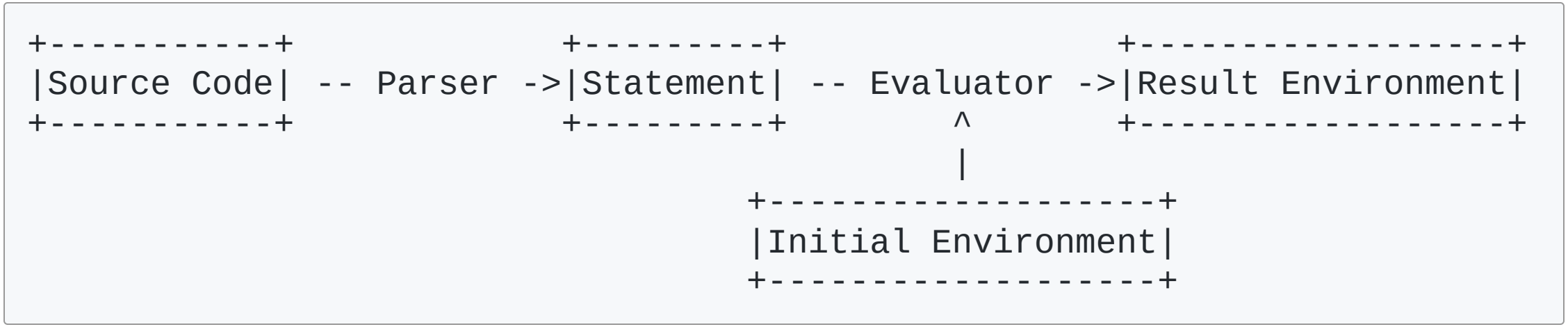
前回までのあらすじ(3)

- 代数データ型を使ってプログラミング言語の構造を表現した(Statement)
- Statement型のデータを評価する関数を作った

次は、パーサ

- パーサ(構文解析器)とは、ソースコードやXMLなど、何らかの言語で記述された構造的な文字データを解析し、プログラムで扱えるようなデータ構造の集合体に変換するプログラムのこと。

つまり、文字列を前回作ったデータ構造(Statement)に変換するプログラム。



新しく登場するHaskellの機能たち

- 型変数
- Maybe型
- caseによるパターンマッチ
- 関数の部分適用
- リストのパターンマッチ
- 文字列型String
- 演算子の定義
- ラムダ関数
- 関数を返す関数

型変数

あるデータ構造をいろんな型に対して適用したいことがある。

```
data IntList = Node Int IntList | IntNil
```

Int以外もリストにしたい、そんなときは型変数。

```
data List a = Node a (List a) | Nil
```

```
let int_list = Node 10 (Node 9 Nil)
let dbl_list = Node 3.14 (Node 2.7 Nil)
```

Maybe型

Maybe型は「失敗するかも知れない」を意味する型

```
data Maybe a = Nothing -- 失敗
             | Just a -- 成功(aには成功した場合の値が入る)
```

```
divine :: Int -> Int -> Maybe Int
divine x y = if y == 0 then Nothing else Just (x `div` y)
```

```
> divine 9 3
Just 3
> divine 9 0
Nothing
```

caseによるパターンマッチ(1)

```
data Shape = Circle Double -- Diameter
           | Rectangle Double Double -- Width, Height
           | Line Double Double -- Angle, Length
```

関数でのパターンマッチ

```
area :: Shape -> Double

area (Circle diameter) = radius * radius * 3.14
    where radius = diameter / 2

area (Rectangle width height) = width * height

area (Line _ _) = 0
```

caseによるパターンマッチ(2)

```
case 変数 of
  パターン1 -> 式
  パターン2 -> 式
  パターン3 -> 式
        :
```

```haskell
area' :: Shape -> Double
area' s = case s of
    Circle diameter -> radius * radius * 3.14
        where radius = diameter / 2
    Rectangle width height -> width * height
    Line _ _ -> 0
```

関数の部分適用

複数の引数をとる関数に対して、一部の引数だけを与えて新しい関数を作ることができる。

```
> add x y = x + y
> add10 = add 10
> add10 20
30
```

```
> message m1 m2 cond = if cond then m1 else m2
> message "ABC" "123" True
"ABC"
> helloOrBye = message "Hello" "Bye"
> helloOrBye False
"Bye"
```

「`=`の左辺を右辺で置き換えられる」ルールを思い出そう。

文字列型String

文字列は`Char`のリスト`type String = [Char]`。リストの`..`記法も使える。

```
> ['a' .. 'z']
"abcdefghijklmnopqrstuvwxyz"
> ['0' .. '9']
"0123456789"
```

文字列はリストなので、リストのパターンマッチが使える。

```
takeHead :: String -> Maybe Char
takeHead [] = Nothing
takeHead (s:_) = Just s
```

文字列はリストなので、文字列の結合は`++`演算子で行う。

```
> "abc" ++ "def"
"abcdef"
```

演算子の定義

- `()`で囲んで、独自の演算子を定義できる。
- `<->`という、結果が0未満にならない引き算演算子を定義してみる

```
(<->) :: Int -> Int -> Int
x <-> y = if r < 0 then 0 else r
    where r = x - y
```

定義した演算子は、他の演算子と同じように使える

```
> 10 <-> 5
5
> 10 <-> 15
0
```

ラムダ関数

無名関数の一種。
Haskellでは関数も他の値と同じように変数に代入したり、関数に渡したりできる。

```
\引数1 引数2 ... 引数n -> 関数本体
```

```
> let f = \x y -> x + y
          --------------- ここがラムダ関数
```

関数を関数の引数として渡すこともできる。

```
> g f x = 2 * (f x)
> g (\x -> x + 1) 9
20
```

関数を返す関数

ラムダ式を使って、関数を返す関数を定義できる。例えば、

```
adderGenerator :: Int -> (Int -> Int)
adderGenerator x = \y -> x + y
```

```
> f = adderGenerator 10
> f 5
15
```

「置き換えられるルール」に従って考えてみると...

```
f 5
-- fの定義より
(adderGenerator 10) 5
-- adderGeneratorの定義より
(\y -> 10 + y) 5
-- 評価すると
15
```

関数型パーサ

関数型パーサの基本的な考え方

小さなパーサを作り、それを組み合わせてより複雑なパーサを作っていく。

例えば、`(8593 - ((3 + 12) * 19))`のような式のパーサを作る場合

- 「特定の１文字を取り出すパーサ char」を作り、
- charを使って「数字１文字を取り出すパーサ digit」と
  「演算子を取り出すパーサ op」を作り、
- digitを使って「複数桁の数字を取り出すパーサ literal」を作り、
- literalとopを使った「式を取り出すパーサ expr」を作る

パーサの型(1)

パーサは関数。文字列を解析し`Statement`を取り出す関数なので、素直に書くと

```
type Parser = String -> Statement
```

しかし、前述のとおり、大きなパーサを作るためには小さなパーサが必要で、それは必ずしも`Statement`を取り出すものではない。なので一般的に書くと

```
type Parser a = String -> a
```

パーサの型(2)

全部の文字列をひとつのパーサで解析するわけではないので、残った文字列は返してもらわないといけない。
例えば、数字を取り出すパーサに"1+2"という文字列を渡したら、1と"+2"を返してほしい。
なので、取り出した結果と残った文字列のタプルを返すようにする。

```
type Parser a = String -> (a, String)
```

パーサの型(3)

そもそも、必ず解析が成功するとは限らない。
例えば、数字を取り出すパーサに"ABCD"という文字列を渡したら、失敗してほしい。

```
type Parser a = String -> Maybe (a, String)
```

- 成功すると`Just (取り出した値, 残りの文字列)`が返ってくる
- 失敗すると`Nothing`が返ってくる

ここから先は難しい話

基礎となる３つのパーサ

```haskell
-- 文字列を消費せず、与えられた値を返す必ず成功するパーサ
ret :: a -> Parser a
ret v = \inp -> Just (v, inp)

-- 文字列を消費せず、必ず失敗するパーサ
failure :: Parser a
failure = \_ -> Nothing

-- 文字列から最初の一文字を取り出すパーサ
-- 文字列が空白なら失敗
item :: Parser Char
item = \inp -> case inp of
    [] -> Nothing
    (x:xs) -> Just (x, xs)
```

パーサを組み合わせる

パーサ同士を組み合わせる、二通りの方法を考える。
ひとつは「そして」演算子 `>>>` 、もうひとつは「または」演算子 `+++` 。

- `>>>` はあるパーサで処理したあと、次のパーサで処理する演算子。
- `+++` はあるパーサで処理してみて、失敗したら別のパーサで処理する演算子。

最難関、「そして」演算子(1)

パーサ`p`と、`p`の解析結果を渡すとパーサを返す関数`g`を取り、
関数`g`が返すのと同じ型のパーサを返す。
この演算子が返すパーサは`p`で解析を試み、

- 失敗すれば、`Nothing`を返す
- 成功すれば、
    i. 解析結果を`g`に渡して、
    ii. `g`が返すパーサで残りの文字列を解析し、
    iii. その解析結果を返す

```
(>>>) :: Parser a -> (a -> Parser b) -> Parser b
p >>> g = \inp -> case p inp of
    Nothing -> Nothing
    Just (r, out) -> p2 out
        where p2 = g r
```

最難関、「そして」演算子(2)

```haskell
(>>>) :: Parser a -> (a -> Parser b) -> Parser b
p >>> g = \inp -> case p inp of
    Nothing -> Nothing
    Just (r, out) -> p2 out
        where p2 = g r
```

Python風に書くと

```python
def then(parser, generator):
    return lambda input:
        (result, out) = parser(input) # parserで文字列を解析
        if result == None && out == None:
            # 解析に失敗したら...
            return (None, None) # このパーサも失敗
        else:
            # parserが解析に成功したら...
            parser2 = generator(result) # 解析結果をパーサ生成器に渡してパーサを取得
            return parser2(out) # 得られたパーサの解析結果を返す
```

最難関、「そして」演算子(3)

これ、何が嬉しいの？
例：数字を消費してその数字を返すパーサ`digit`と、ドットを消費して"."を返すパーサ`dot`があるとする。

`digit "123abc"` → `("123","abc")`
`dot ".abc"` → `(".","abc")`

値を渡されると「文字列を消費せず、渡された値を返すパーサ」を返す関数を作る。

```
gen1 x = ret x
gen1 = \x -> ret x
```

では、これは？

```
digit >>> gen1
```

最難関、「そして」演算子(4)

`p >>> g`は、パーサ`p`と、`p`の解析結果を渡すとパーサを返す関数`g`を取り、
関数`g`が返すのと同じ型のパーサを返す。
この演算子が返すパーサは`p`で解析を試み、

- 失敗すれば、`Nothing`を返す
- 成功すれば、
    i. 解析結果を`g`に渡して、
    ii. `g`が返すパーサで残りの文字列を解析し、
    iii. その解析結果を返す

```
gen1 = \x -> ret x
digit >>> gen1
```

これは、数字を消費してその数字を返すパーサ。

最難関、「そして」演算子(5)

```
gen1 = \x -> ret x
digit >>> gen1
```

`gen1` の定義を展開して

```
digit >>> (\x -> ret x)
```

数字を消費してその数字を返すパーサ。

最難関、「そして」演算子(6)

次に以下のパーサ生成関数を考える

```
gen2 = \x1 -> (digit >>> (\x2 -> ret (x1 ++ x2)))
```

これは、文字列を受け取ると「数字を消費して、受け取った文字列とその数字を結合して返すパーサ」を返す関数。

では、これは？

```
gen2 = \x1 -> (digit >>> (\x2 -> ret (x1 ++ x2)))
dot >>> gen2
```

これは、ドットを消費して「数字を消費して、受け取った文字列(=ドット)とその数字を結合して返す」パーサ

最難関、「そして」演算子(7)

```
gen2 = \x1 -> (digit >>> (\x2 -> ret (x1 ++ x2)))
dot >>> gen2
```

`gen2` の定義を展開して

```
dot >>> (\x1 -> (digit >>> (\x2 -> ret (x1 ++ x2))))
```

これは、".123"や".049"といった1未満の小数を解析できるパーサになっている。

最難関、「そして」演算子(8)

```
dot >>> (\x1 -> (digit >>> (\x2 -> ret (x1 ++ x2))))
```

実は、`>>>`や`->`の結合規則から、カッコがなくても正しく解釈される。
ついでに名前もつけておく(Float less than 1)

```
fload_lt_1 = dot >>> \x1 -> digit >>> \x2 -> ret (x1 ++ x2)
```

適切に改行すると...

```
float_lt_1 = dot >>> \x1 -- ドットを取得してx1に入れ
           -> digit >>> \x2 -- 数字を取得してx2に入れ
           -> ret (x1 ++ x2) -- その二つを結合したものを返す(ret)
```

複数のパーサを順番に適用して、最後にひとまとめにして`ret`で返す。

「または」演算子

`p`と`q`２つのパーサを取り、同じ型のパーサを返す。
この演算子が返すパーサは

- `p`で解析を試み、成功すればその結果を返す
- 失敗すれば`q`で解析した結果を返す
- (両方失敗した場合は`Nothing`が返ることになる)

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case p inp of
    Nothing -> q inp
    Just (result, out) -> Just (result, out)
```

`>>>`と`+++`を使って、多彩なパーサを組み上げていく。

難しい話はここまで

条件を満たす一文字を取り出すパーサ

```
sat :: (Char -> Bool) -> Parser Char
sat p = item >>> \x
     -> if p x then ret x else failure
```

```
> sat (\c -> c == 'A') "ABC"
Just ('A',"BC")
> sat (\c -> c == 'A') "123"
Nothing
```

アルファベットや数字、特定の文字を取り出すパーサ

```
isIn :: Char -> [Char] -> Bool
isIn _ [] = False
isIn t (c:cs) = if c == t then True else isIn t cs

isDigit x = isIn x ['0'..'9']
isAlpha x = isIn x (['a'..'z'] ++ ['A'..'Z'])

digit = sat isDigit
alpha = sat isAlpha
char x = sat (\y -> x==y)
```

```
> digit "123"
Just ('1',"23")
> alpha "abc"
Just ('a',"bc")
> char '-' "-1"
Just ('-',"1")
```

特定の文字列を取り出すパーサ

再帰的なパーサ。

```
string :: String -> Parser String
string [] = ret []
string (x:xs) = char x >>> \_
              -> string xs >>> \_
              -> ret (x:xs)
```

```
> string "while" "while (1)"
Just ("while"," (1)")
> string "while" "if (1)"
Nothing
```

パーサを繰り返し適用するパーサ

あるパーサを失敗するまで繰り返し適用し、成功した結果をリストにして返す。
`many`は0回以上、`many1`は1回以上適用が成功することを要求する。

```
many :: Parser a -> Parser [a]
many p = many1 p +++ ret []

many1 :: Parser a -> Parser [a]
many1 p = p >>> \v
        -> many p >>> \vs
        -> ret ([v] ++ vs)
```

```
> (many digit) "123abv"
Just ("123","abv")
> (many1 digit) "123abv"
Just ("123","abv")
> (many digit) "abc"
Just ("","abc")
> (many1 digit) "abc"
Nothing
```

前後の空白を無視するパーサ

`p`を適用する前後に`space`(スペース、タブ、改行を取り出すパーサ)を適用して、その結果は無視する。

```
isSpace x = isIn x (" \t\n")
space = sat isSpace

token :: Parser a -> Parser a
token p = (many space) >>> \_
        -> p >>> \x
        -> (many space) >>> \_
        -> ret x
```

```
> (token (many1 digit)) "  123 + 456"
Just ("123","+ 456")
```

これまでに用意した部品

- `>>>`：「そして」演算子。複数のパーサを連結し、その結果をまとめて返す
- `+++`：「または」演算子。複数のパーサを順に適用し最初に成功したものを返す
- `sat f`：条件を満たす一文字を取り出す
- `char c`：特定の一文字を取り出す
- `digit`：数字を一文字を取り出す
- `alpha`：アルファベットを一文字を取り出す
- `string s`：特定の文字列を取り出す
- `many p`：`p`が失敗するまで0回以上`p`を繰り返し適用し、結果をリストにする
- `many1 p`：`p`が失敗するまで1回以上`p`を繰り返し適用し、結果をリストにする
- `token p`：前後の空白を無視して`p`を適用する

これで、プログラミング言語パーサに必要な一通りの部品が揃った。

パースして作りたいデータ

```
data BinaryOperator
    = Add | Sub | Mul | Div
    | Lt | Le | Gt | Ge | Eq | Neq
    | And | Or

data Expression
    = Literal BaseType
    | Variable VariableName
    | BinaryOperation BinaryOperator Expression Expression

data Statement
    = DeclareVariable VariableName Expression
    | Sequence Statement Statement
    | Assign VariableName Expression
    | If Expression Statement Statement
    | Pass
    | While Expression Statement
```

Literal

```
literal :: Parser Expression
literal = many1 digit >>> \x
        -> ret (Literal (read x))
```

```
> literal "123+456"
Just (Literal 123,"+456")
> literal "x+1"
Nothing
```

Variable

```
identifier :: Parser String
identifier = (char '_') +++ alpha >>> \h
           -> many ((char '_') +++ alpha +++ digit) >>> \t
           -> ret ([h] ++ t)

variable :: Parser Expression
variable = identifier >>> \v
         -> ret (Variable v)
```

```
> variable "x+1"
Just (Variable "x","+1")
> variable "_12+1"
Just (Variable "_12","+1")
> variable "2+3"
Nothing
```

BinaryOperation (1)

```
expression :: Parser Expression
expression = literal +++ variable +++ binaryOperation

binaryOperator :: Parser BinaryOperator
binaryOperator = add +++ sub +++ mul +++ div +++ lt +++ le +++ gt +++ ge +++ eq +++ neq +++ and +++ or
    where
        add = string "+" >>> \_ -> ret Add
        sub = string "-" >>> \_ -> ret Sub
        mul = string "*" >>> \_ -> ret Mul
        div = string "/" >>> \_ -> ret Div
        lt = string "<" >>> \_ -> ret Lt
        le = string "<=" >>> \_ -> ret Le
        gt = string ">" >>> \_ -> ret Gt
        ge = string ">=" >>> \_ -> ret Ge
        eq = string "==" >>> \_ -> ret Eq
        neq = string "!=" >>> \_ -> ret Neq
        and = string "&&" >>> \_ -> ret And
        or = string "||" >>> \_ -> ret Or

binaryOperation :: Parser Expression
binaryOperation = token (char '(') >>> \_
               -> token expression >>> \ls
               -> token binaryOperator >>> \op
               -> token expression >>> \rs
               -> token (char ')') >>> \_
               -> ret (BinaryOperation op ls rs)
```

BinaryOperation (2)

```
> binaryOperation "(1 + 2)"
Just (BinaryOperation Add (Literal 1) (Literal 2),"")

> binaryOperation "(1 + va)"
Just (BinaryOperation Add (Literal 1) (Variable "va"),"")

> binaryOperation "(1 + x)"
Just (BinaryOperation Add (Literal 1) (Variable "x"),"")

> binaryOperation "(1 + (2 && x_1))"
Just (BinaryOperation Add (Literal 1) (BinaryOperation And (Literal 2) (Variable "x_1")),"")

*Parser> binaryOperation "(1 + (2 + ))"
Nothing

*Parser> binaryOperation "(1 + (2 + - 33 ))"
Nothing
```

試しに ((1+2)*3)を計算してみる

```
> expression "((1 + 2) * 3)"
Just (BinaryOperation Mul (BinaryOperation Add (Literal 1) (Literal 2)) (Literal 3),"")
```

パーサの返すデータの型は`Maybe (a, String)`ほしいのは`a`の部分だけなので、パターンマッチで取り出す。

```
> Just (e, _) = expression "((1 + 2) * 3)"
> e
BinaryOperation Mul (BinaryOperation Add (Literal 1) (Literal 2)) (Literal 3)
```

今回は特に環境(Environment)を必要としないので(式の中で変数を使っていない)、環境として`[]`を渡して評価する。

```
> expressionEvaluation [] e
9
```

DeclareVariable

```
var 変数名 := 式
```

```
data Statement
    = DeclareVariable VariableName Expression
    (略)
```

```
declareVariable :: Parser Statement
declareVariable = token (string "var") >>> \_
                -> token identifier >>> \v
                -> token (string ":=") >>> \_
                -> token expression >>> \init
                -> ret (DeclareVariable v init)
```

```
> declareVariable "    var x := 10;"
Just (DeclareVariable "x" (Literal 10),";")
```

Assign

```
変数名 := 式
```

```
data Statement
    (中略)
    | Assign VariableName Expression
    (略)
```

```
assign :: Parser Statement
assign = token identifier >>> \v
      -> token (string ":=") >>> \_
      -> token expression >>> \e
      -> ret (Assign v e)
```

If

```
if 条件式 { 文1 } else { 文2 }
```

```
data Statement
    (中略)
    | If Expression Statement Statement
    (略)
```

```
if_ :: Parser Statement
if_ = token (string "if") >>> \_
   -> token (expression) >>> \cond
   -> token (string "{") >>> \_
   -> token (statement) >>> \thenS
   -> token (string "}") >>> \_
   -> token (string "else") >>> \_
   -> token (string "{") >>> \_
   -> token (statement) >>> \elseS
   -> token (string "}") >>> \_
   -> ret (If cond thenS elseS)
```

Pass

```
pass
```

```
data Statement
    (中略)
    | Pass
    (略)
```

```
pass :: Parser Statement
pass = token (string "pass") >>> \_
    -> ret Pass
```

While

```
while 条件式 { 文 }
```

```
data Statement
    (中略)
    | While Expression Statement
```

```
while :: Parser Statement
while = token (string "while") >>> \_
     -> token (expression) >>> \cond
     -> token (string "{") >>> \_
     -> token (statement) >>> \s
     -> token (string "}") >>> \_
     -> ret (While cond s)
```

Sequence (1)

```
data Statement
    (中略)
    | Sequence Statement Statement
    (略)
```

連続した文をSequenceとして解釈する場合、複数通りの解釈があり得る。
例えば、`A; B; C; D`という４つの文からなるStatementの場合

```
Sequence A (Sequence B (Sequence C D))
Sequence (Sequence A B) (Sequence C D)
Sequence (Sequence (Sequence A B) C) D
```

などのバリエーションが考えられる。

Sequence (2)

ひとつめのStatementにSequenceが入らないようにすることで、これを回避する。
(前の例で言えば、`Sequence A (Sequence B (Sequence C D))`に固定する)

```
statement :: Parser Statement
statement = Parser.sequence +++ withoutSequence

withoutSequence :: Parser Statement
withoutSequence = declareVariable +++ assign +++ if_ +++ pass +++ while

sequence :: Parser Statement
sequence = token withoutSequence >>> \s1
         -> token (string ";") >>> \_
         -> token statement >>> \s2
         -> ret (Sequence s1 s2)
```

動かしてみる

xに代入された値の平方根を求めるスクリプト。

$$a^2 \leqq x < b^2$$

```
var x := 917384;
var a := 0;
var b := (x + 1);
var m := 0;
while ((a + 1) != b) {
  m := ((a + b) / 2);
  if (x >= (m * m)) {
    a := m
  }
  else {
    b := m
  }
};
var veri1 := (a * a);
var veri2 := (b * b)
```

動かしてみる

実行結果

```
veri2   : 917764
veri1   : 915849
m       : 957
b       : 958
a       : 957
x       : 917384
```

まとめ

- 基礎となるパーサ `ret`、 `failure`、 `item`
- 基礎となる演算子 `>>>` (そして)、 `+++` (または)
- これらを組み合わせて複雑な構文を解析するパーサを実現できる
- パース結果を前回実装した評価関数に渡せば、インタプリンタを実装できる
- 平方根を求めるスクリプトで動作を確認した